CMSC818B
HW-1

Rachith Prakash
116141468

Problem 1:

Few notes:

- Here, we are given inputs as list of vertices in the format (vertex id, x-coordinate, y-coordinate).
- Also, the input file says that the edge weight is euclidean, hence these coordinates can be viewed/manipulated in euclidean space.
- Vertex has the following properties - key, id, parent, child, x-coordinate, y-coordinate
- Here key is the length of walk from its parent to itself.
- Edge has the following properties - weight, start-vertex, end-vertex.
- While reading the file, all vertex will be updated with respective id, x-coordinate, y-coordinate, parent=null, key=inf.
- A custom queue (used vector in C++) wrapper is written which contains an array of pointers pointing to these vertices and sorted through their key value (In decreasing order).
- The graph is assumed to be complete. Hence, all vertices are connected to every other vertex.

Implementation:

MST computation:

- A random node is chosen.
- It’s key is set to zero. It’s parent is still null.
- MST is a vector of pointers to vertices. Currently it is null i.e. does not have any elements.
- Q is the custom queue. It contains all vertices in sorted order
- **Pseudo algorithm**

```
While (Q is not empty)
    Add top vertex to MST (top vertex == vertex with lowest key)
    Pop top vertex from Q
    For Every other vertex in Q
        If (vertex not in MST) and (weight of edge to popped vertex < current vertex key)
            Elements parent = popped element
            Elements key = weight of edge to popeed vertex
```

- Once the MST if formed, all the vertices are updated to point to its children.
- Using this MST, TSP tour is performed using pre-order traversal. This would add shortcuts while performing DFS. We can see that this length is < 2*optimal-length
- Once TSP tour is obtained, a graph and plotted to see it visually.
- This shows that several edges cross each other making it an inefficient tour.
- I apply a heuristic on this tour to unravel the edge crossings.
- **Pseudo algorithm**

```
While (tour has not no edge crossings)
    For all edges in TSP tour
        For all edges in TSP tour starting from edges+2
            If (Is any edge crossing in TSP tour)
                Unravel those edges
```

- For checking edge crossing, given two distinct edges, 4 vertices can be extracted. From these points, point of intersection can be extracted.
- Once point of intersection is obtained, check if it is enclosed by these points. If it does, then edge crossing is true.

Observations:

- The MST remains the same with change in root node. However, different root nodes produce different TSP tour lengths.
- My tour cost does consider the last edge's (last node to the root node) weight.
- Different runs takes different node as the root node, thus the optimal length potentially changes with change in the root node. This also changes the time taken to execution.
- Given below is one such instance with root node being different.

File name	MST length	Optimal tour length found in TSPLIB documentation	TSP tour length without applying heuristic	TSP tour length using heuristics	Time taken to run in seconds (without visualization of graph)
eil51	376.491	426	580.695	498.857	0.0033
eil76	472.331	538	725.568	657.314	0.0078
eil101	562.257	629	893.15	793.798	0.016

The same algorithm explained above is implemented for random inputs of 100,200,300 with 10 iterations each and output is shown below.

10 random instances of 100 nodes each.

Sl. No.	MST length	TSP tour length without applying heuristic	TSP tour length using heuristics	Time taken to run in seconds without visualization
1	673.304	1089.14	985.62	0.0137
2	656.54	1056.46	954.88	0.0147
3	659.5	1061.89	1010.93	0.014
4	680.42	1094.70	1031.67	0.0133
5	666.24	1100.22	1022.895	0.0129
6	680.76	1059.48	991.85	0.0131
7	702.87	1148.45	1012.96	0.0130
8	706.58	1205.33	1069..26	0.0138
9	624.90	993.884	892.98	0.0132
10	617.46	988.69	914.80	0.013

10 random instances of 200 Nodes each.

Sl. No.	MST length	TSP tour length without applying heuristic	TSP tour length using heuristics	Time taken to run in seconds
1	1856.72	3006.82	2709.98	0.0594
2	1877.82	2969.75	2696.211	0.06
3	1856.688	3032.75	2680.52	0.061
4	1947.25	3093.98	2918.82	0.059
5	1814.224	2847.27	2637.03	0.059
6	1893.22	3015.81	2713.92	0.063
7	1820.5	2926.99	2732.24	0.059
8	1829.54	2849.94	2626.44	0.058
9	1967.68	3094.70	2679.53	0.058
10	1931.64	2994.31	2718.6	0.062

10 random instances of 300 Nodes each

Sl. No.	MST length	TSP tour length without applying heuristic	TSP tour length using heuristics	Time taken to run in seconds
1	3394.11	5519.47	5138.3	0.147
2	3471.60	5556.01	4967.23	0.151
3	3403.055	5344.72	4904.76	0.147
4	3430.925	5544.16	4895.11	0.144
5	3397.227	5246.45	4891.84	0.144
6	3522.92	5648.7	5087.35	0.151
7	3453.79	5335.26	4742.90	0.146
8	3499.44	5683.45	5096.18	0.144
9	3521.46	5668.98	5118.26	0.145
10	3395.87	5252.7	4855.34	0.143

Screencast link:
https://drive.google.com/file/d/1Fd1FZrSR9mWm-g1-DHqY-uyGQODymNJe/view?usp=sharing

Problem 2:

This problem uses code from previous problem to solve for TSP. I'll call it TSP() in my pseudo code below. The input for TSP() would be vector of pointers to vertices. Output would be a length of the TSP tour with Heuristic applied.

Q_not_in_tour is a vector whose elements are pointers to vertices in graph. This is to keep track of visited nodes in union with all tours. Vertices can be in any order inside this vector.
k - number of robots/tours
i_tour - ith tour
s - pointer to root node

Add s to all the k tours. Hence, initially all tours will have one element, ie. root node. TSP tour of all the tours would be the same.

Pseudo code-

```
For all elements, e in Q_not_in_tour
    min_length = 0
    min_index = 0
    For j in all tours
        min_length = min(min_length, TSP(j_tour + e))
        min_index = j
    {min_index}_tour.add(e)
```

I’m going through all vertices and adding them to a tour that upon addition with this node has the least length with respect to the length obtained by adding the same node to other tours. By this method I will choose tours that have least length everytime I’m adding a node. This will ensure that all nodes are having approximately comparable lengths by the end.

Time complexity of TSP is O(n2), where n is the number of nodes in the graph. From the above pseudo code, the time complexity is given by O(k*n3). K is the number of tours, n is cubed. Hence, for large graphs and large number of tours this algorithm performs bad.

A simple case when algorithm is bad is k=1. Ideally it should be O(n2) with just one TSP running. But, this algorithm performs in O(n3) which is a poor performance.